Assignment 2

Name: Haopeng Chen

ID: 120090645

How to Start

```
# Build
cmake -B build
cmake --build build -j 8

# Test for [1024x1024]
sh src/sbach.sh

# Test for [2048x2048]
sh src/sbatch.sh
```

Performance Overview

Methods	Matrix 1024x1024	Matrix 2048x2048
Naïve	8143	102244
Locality	383	3100
SIMD+Locality	89	728

On **Matrix 1024x1024**

Methods \ Threads	1	2	4	8	16	32
OpenMP + SIMD + Memory Locality	88	83	54	41	29	27
Methods \ Processes x Threads	**1 x 32**	**2 x 16**	**4 x 8**	**8 x 4**	**16 x 2**	**32 x 1**
MPI + OpenMP + SIMD + Memory Locality	58	34	20	20	16	20

On **Matrix 2048x2048**

Methods \ Threads	1	2	4	8	16	32
OpenMP + SIMD + Memory Locality	703	663	350	184	114	88
Methods \ Processes x Threads	**1 x 32**	**2 x 16**	**4 x 8**	**8 x 4**	**16 x 2**	**32 x 1**
MPI + OpenMP + SIMD + Memory Locality	130	114	83	92	79	96

Analysis

On **Matrix 2048x2048**

Methods	CPU-Cycles	Cache-Misses	Page-Faults
Naïve	427k	414k	1k
Locality	16k	15k	723
SIMD+Locality	6k	6k	81

We notice that under single process single thread situation, improvement of *cache-miss* and *page-fault* could reduce CPU executions cycles, that is, boost the running speed. For details, it is the technique of how computers load data. We know that data is loaded **from memory to registers**, through a hierarchical of caches. The key point is that we need to **reduce the waiting cycles** of CPU for data loading, a.k.a. try to **use data in cache** instead of loading it again and again from memory. Recall the knowledge in OS, data in memory and cache is **organized into pages** for convenience. When we need to load data, the whole page will be load into next level cache, **with the our destinated variable and its consecutive ones**. Our optimization is trying to use these consecutive variables successively.

For this matrix multiplication, we notice that **scalers in one row are consecutive** while **rows are discrete**. However, the *Naïve* implementation accesses these scalers in the sequence of columns, which leads to **successive cross-rows access**, resulting in time waste in CPU waiting cycles.

We optimize access sequence, eliminating cross-rows accesses to least frequently. That is, during the inner-most loop, we do **not have any cross-rows access but only the same row accesses** successively. Also, we **unroll the loop in multiples of 4**, trying to maximize the usage of caches. As sizes of most caches are multiples of 16, we believe that reading 4 `int32` in one loop could **sufficiently utilize caches** and boost up the performance. After employing these two optimizations, we observe that the cache-miss drops dramatically, as well as decrease in page-fault. These prove the efficiency of our methods.

When it comes into SIMD, it introduces **parallelism** into our optimizations, as it could **combine several identical type of operations into one CPU cycle**. We move our step from CPU waiting cycles into **CPU execution cycles**. Previous implementations execute only one multiplication in one CPU cycle, while we could combine 8 of them into one with AVX2 support. This is also beneficial to avoid cache-miss and page-fault. As we always accessing consecutive data and write-back them consecutively, we could use caches better.

OpenMP+SIMD+Locality (Processes x Threads)	CPU-Cycles	Cache-Misses	Page-Faults
1	6k	6k	85
2	8k	7k	82
4	9k	7k	160
8	9k	7k	197
16	10k	6k	223

OpenMP+SIMD+Locality (Processes x Threads)	CPU-Cycles	Cache-Misses	Page-Faults
32	12k	6k	240

PS: We do not list MPI here, since multiprocesses leads to chaos in output terminal.

Multithreads and multiprocesses are identical underneath. They do **not boost by reducing average efficient execution time of one operation** (execution + waiting time), but utilize multiple ALUs / Cores to finish work simultaneously. Its key point is that by using more ALUs / Cores, we could **reduce the total work on each one**, giving less gross time usage. We also find that although multithreads / multiprocesses model could boost up the gross performance, it harms memory / caches access. According to the table above, we find that cache-miss and page-fault increase as threads / processes goes up. We believe that it is because different workers (threads / processes) try to accesses various rows in a short time, making the content in cache substituted very quickly, even it is needed later.